Mokhov A, Carmona J, Beaumont J.

Mining Conditional Partial Order Graphs from Event Logs.

***LNCS Transactions on Petri Nets and Other Models of Concurrency* 2016, 9930, 114-136.**

Mining Conditional Partial Order Graphs from Event Logs

Andrey Mokhov[1], Josep Carmona[2], Jonathan Beaumont[1]

[1] Newcastle University, United Kingdom
andrey.mokhov@ncl.ac.uk, j.r.beaumont@ncl.ac.uk
[2] Universitat Politècnica de Catalunya, Spain
jcarmona@cs.upc.edu

Abstract Process mining techniques rely on *event logs*: the extraction of a process model (*discovery*) takes an event log as the input, the adequacy of a process model (*conformance*) is checked against an event log, and the *enhancement* of a process model is performed by using available data in the log. Several notations and formalisms for event log representation have been proposed in the recent years to enable efficient algorithms for the aforementioned process mining problems. In this paper we show how *Conditional Partial Order Graphs* (CPOGs), a recently introduced formalism for compact representation of families of partial orders, can be used in the process mining field, in particular for addressing the problem of compact and easy-to-comprehend representation of event logs with data. We present algorithms for extracting both the control flow as well as the relevant data parameters from a given event log and show how CPOGs can be used for efficient and effective visualisation of the obtained results. We demonstrate that the resulting representation can be used to reveal the hidden interplay between the control and data flows of a process, thereby opening way for new process mining techniques capable of exploiting this interplay. Finally, we present open-source software support and discuss current limitations of the proposed approach.

1 Introduction

Event logs are ubiquitous sources of process information that enabled the rise of the *process mining* field, which stands at the interface between formal methods, concurrency theory, machine learning, and data visualisation [1]. A *process* is a central notion in process mining and in computing science in general, and the ability to automatically discover and analyse evidence-based process models is of utmost importance for many government and business organisations. Furthermore, this ability is gradually becoming a necessity as the digital revolution marches forward and traditional process analysis techniques based on the explicit construction of precise process models are no longer adequate for continuously evolving large-scale real-life processes, because our understanding of them is often incomplete and/or inconsistent.

At present, the process mining field is mainly focused on three research directions: i) the *discovery* of a process model, typically, a Petri Net or a BPMN (Business Process Model and Notation); ii) the *conformance* analysis of a process

model with respect to a given event log; and iii) the *enhancement* of a process model using additional information (i.e., *data*) contained in an event log. The bulk of research in these directions has been dedicated to the design of the algorithmic foundation and associated software tools with many notable successes, such as, e.g. the ProM framework [2].

However, a more basic problem of *event log representation and visualisation* received little attention to date, despite the fact that effective visualisation is essential for achieving a good understanding of the information contained in an event log. Indeed, even basic *dotted charts* prove very useful for describing many aspects of event logs even though they are just simple views of event log traces plotted over time [3].

In this paper we discuss the application of *Conditional Partial Order Graphs* (CPOGs) for event log representation and visualisation. The CPOG model has been introduced in [4] as a compact graph-based formalism for complex concurrent systems, whose behaviour could be thought of as a collection of multiple partial order scenarios (see a formal definition in §4). The key idea behind our approach is to convert a given event log into a collection of partial orders, which can then be compactly described and visualised as a CPOG, as explained in the motivating example in §2. CPOGs are less expressive than Petri Nets and have important limitations, such as the inability to represent cyclic behaviour, but they are well-suited for representing inherently acyclic event logs.

We see CPOGs not as the end product of process mining, but as a convenient intermediate representation of event logs that provides much better clarity of visualisation as well as better compactness, which is important for the efficiency of algorithms further in the process mining pipeline. Furthermore, CPOGs can be manipulated using algorithmically efficient operations such as *overlay* (combining several event logs into one), *projection* (extracting a subset of interesting traces from an event log), *equivalence checking* (verifying if two event logs describe the same behaviour) and others, as formalised in [5].

The contributions of this paper[3] are:

- We propose two methods for mining compact CPOG representations from event logs, see §5. The methods are based on the previous research in CPOG synthesis [4], and on a novel concurrency oracle introduced in §5.2.
- We propose techniques for extracting data parameters from the information typically contained in event labels of a log and for using these parameters for annotating derived CPOG models, thereby providing a direct link between the control and data aspects of a system under observation, see §6.
- We present an opensource implementation of the CPOG mining methods as a Workcraft plugin [7] and as a command line tool PGminer [8], see §7.
- We evaluate our implementation on several event logs known to the process mining community, see §7.3. The experimental results show that the current implementation is capable of handling large real-life logs in reasonable time and highlight the areas where future research work is needed. We review and discuss related work in §8.

[3] This paper is an extended version of [6].

2 Motivating Example

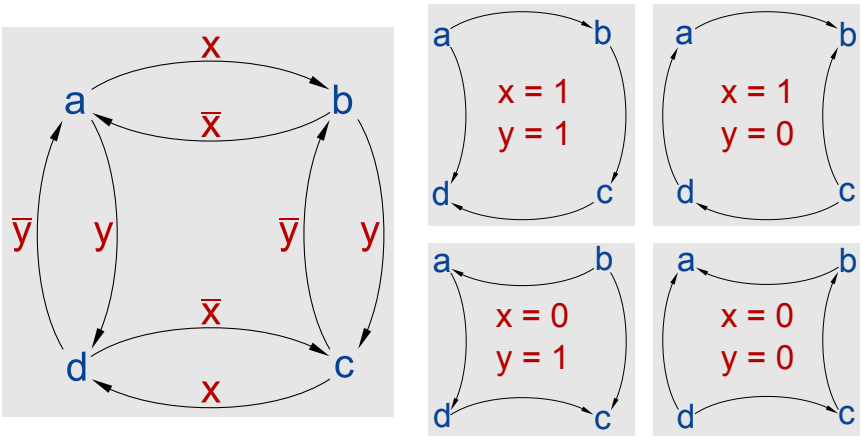
We start by illustrating the reasons that motivate us to study the application of CPOGs in process mining, namely: (i) the ability of CPOGs to compactly represent complex event logs and clearly illustrate their high-level properties, and (ii) the possibility of capturing event log meta data as part of a CPOG representation, thereby taking advantage of the meta data for the purpose of explaining the process under observation.

Consider an event log $L = \{abcd, cdab, badc, dcba\}$. One can notice that the order between events a and b always coincides with the order between events c and d. This is an important piece of information about the process, which however may not be immediately obvious when looking at the log in the text form. To visualise the log one may attempt to use existing process mining techniques and discover a graphical representation for the log, for example in the form of a Petri Net or a BPMN. However, an exact Petri Net representation of event log L is cumbersome and difficult to understand. Furthermore, existing Petri Net based process mining techniques perform very poorly on this log. To compare the models discovered from this log by several popular process mining methods, we will describe the discovered behaviour by regular expressions, where operators $||$ and $\cup$ denote interleaving and union, respectively.

The α-algorithm [9] applied to L produces a Petri Net accepting the behaviour $a \cup b \cup c \cup d$, which clearly cannot reproduce any of the traces in L. Methods aimed at deriving block-structured process models [10][11] produce a connected Petri Net that with the help of *silent* transitions reproduces the behaviour $a \,||\, b \,||\, c \,||\, d$, which is a very imprecise model accepting all possible interleavings of the four events. The region-based techniques [12] discover the same behaviour as the block-structured miners, but the derived models are not connected. One can use classical synthesis techniques to exclude wrong continuations (such as $acbd$, $acdb$, etc.), from the resulting Petri Net [13], however, this process is hard to automate and still leads to inadequately complex models.

CPOGs, however, can represent L exactly and in a very compact form, as shown in Fig. 1(a). Informally, a CPOG is an overlay of several partial orders that can be extracted from it by assigning values to variables that appear in the conditions of the CPOG vertices and arcs. For example, the upper-left graph shown in Fig. 1(b) (assignment $x = 1$, $y = 1$) corresponds to the partial order containing the causalities $a \prec b$, $a \prec d$, $b \prec c$, $c \prec d$. One can easily verify that the model is precise by trying all possible assignments of variables x and y and checking that they generate the traces $\{abcd, cdab, badc, dcba\}$ as desired, and nothing else. See Fig. 1(b) for the corresponding illustration. The compactness of the CPOG representation is due to the fact that several event orderings are overlayed on top of each other taking advantage of the similarities between them. See §4 and §5 for a formal introduction to CPOGs and synthesis algorithms that can be used for mining CPOGs from event logs.

It is worth mentioning that CPOGs allow us to recognise *second-order relations* between events. These are relations that are not relating events themselves, but are relating relations between events: indeed, the CPOG in Fig. 1(a) clearly



(a) CPOG representation (b) Four projections of the CPOG

Figure 1: Exact CPOG representation of log $L = \{abcd, cdab, badc, dcba\}$

shows that the relation between a and b is equal to the relation between c and d, and the same holds for pairs (a, d) and (b, c). In principle, one can go even further and consider third-order relations and so forth. The practical use of such a relation hierarchy is that it may help to extract an event hierarchy from event logs, thereby simplifying the resulting representation even further.

One may be unsatisfied by the CPOG representation in Fig. 1(a) due to the use of 'artificial' variables x and y. Where do these variables come from and what exactly do they correspond to in the process? We found out that additional data which is often present in event logs can be used to answer such questions. In fact, as we will show in §6, it may be possible to use easy-to-understand predicates constructed from the data instead of 'opaque' Boolean variables.

For example, consider the same event log L but augmented with temperature data attached to the traces:

- $abcd, t = 25°$
- $cdab, t = 30°$
- $badc, t = 22°$
- $dcba, t = 23°$

With this information at hand we can now explain what variable x means. In other words, we can open the previously opaque variable x by expressing it as a predicate on temperature t:

$$x = t \geq 25°$$

One can subsequently drop x completely from the CPOG by using conditions $t \geq 25°$ and $t < 25°$ in place of x and $\overline{x}$, respectively, as shown in Fig. 2.

In summary, we believe that CPOGs bring unique event log visualisation capabilities to the process mining field. One can use CPOGs as an intermediate representation of event logs, which can be exact as well as more comprehensible both for humans and for software tools further in the process mining pipeline.

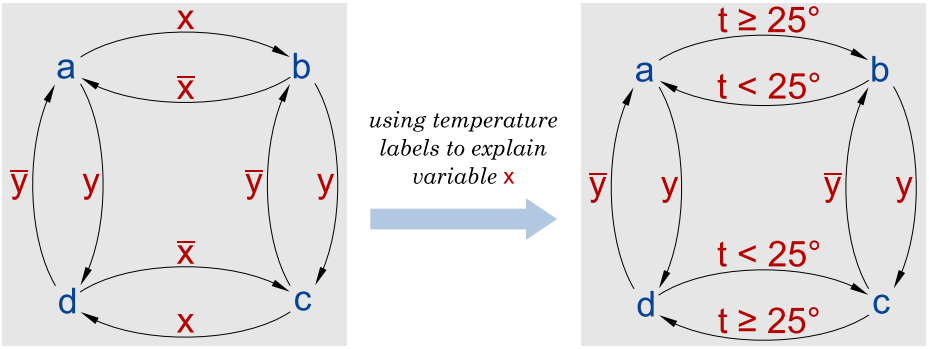


Figure 2: Using event log data to explain CPOG variables

3 Event Logs

In this section we introduce the notion of an *event log*, which is central for this paper and for the process mining field. We also discuss important quality metrics that are typically used to compare methods for event log based process mining.

Table 1 shows a simple event log, which contains not only event information but also *data* in the form of *event attributes*. The underlying traces of the log are $\{abcd, cdab, badc, dcba\}$, just as in the previous section, and they correspond to 'case IDs' 1, 2, 3 and 4, respectively. We assume that the set of attributes is fixed and the function *attr* maps pairs of events and attribute names to the corresponding values. For each *event* e the log contains the case ID $case(e)$, the activity name $act(e)$, and a set of attributes, e.g. $attr(e, timestamp)$. As an example, $case(e_7) = 2$, $act(e_7) = a$, $attr(e_7, timestamp) =$ "10-04-2015 10:28pm", and $attr(e_7, cost) = 19$ in Table 1. Given a set of events E, an *event log* is a multiset of *traces* E^* of events, where events are identified by the activities *act*.

Process mining techniques use *event logs* containing footprints of real process executions for discovering, analysing and extending formal process models, which reveal real processes in a system [1]. The process mining field has risen around a decade ago, and since then it has evolved in several directions, with process discovery being perhaps the most difficult challenge, as demonstrated by a large number of existing techniques. Discovered process models are typically ranked across the following quality metrics, some of which are mutually exclusive:

- *fitness*: the ability of the model to reproduce the traces in the event log (in other words, not too many traces are lost);
- *precision* of the representation of the event log by the model (the opposite of fitness, i.e. not too many new traces are introduced);
- *generalisation*: the ability of the model to generalise the behaviour covered by the event log;
- *simplicity*: the *Occam's Razor* principle that advocates for simpler models.

We present new methods for CPOG mining from event logs and analyse their performance. A qualitative study with respect to the above metrics is beyond the scope of this paper and is left for future research.

Event	Case ID	Activity	Timestamp	Temperature	Cost	Risk
1	1	a	10-04-2015 9:08am	25.0	17	Low
2	2	c	10-04-2015 10:03am	28.7	29	Low
3	2	d	10-04-2015 11:32am	29.8	16	Medium
4	1	b	10-04-2015 2:01pm	25.5	15	Low
5	1	c	10-04-2015 7:06pm	25.7	14	Low
6	1	d	10-04-2015 9:08pm	25.3	17	Medium
7	2	a	10-04-2015 10:28pm	30.0	19	Low
8	2	b	10-04-2015 10:40pm	29.5	22	Low
9	3	b	11-04-2015 9:08am	22.5	31	High
10	4	d	11-04-2015 10:03am	22.0	33	High
11	4	c	11-04-2015 11:32am	23.2	35	High
12	3	a	11-04-2015 2:01pm	23.5	40	Medium
13	3	d	11-04-2015 7:06pm	28.8	43	High
14	3	c	11-04-2015 9:08pm	22.9	45	Medium
15	4	b	11-04-2015 10:28pm	23.0	50	High
16	4	a	11-04-2015 10:40pm	23.1	35	Medium

Table 1: An example event log

4 Conditional Partial Order Graphs

Conditional Partial Order Graphs (CPOGs) were introduced for the compact specification of concurrent systems comprised from multiple behavioural scenarios [4]. CPOGs are particularly effective when scenarios of the system share common patterns, which can be exploited for the automated derivation of a compact combined representation of the system's behaviour. CPOGs have been used for the design of asynchronous circuits [14] and processor microcontrollers [15]. In this paper we demonstrate how CPOGs can be employed in process mining.

4.1 Basic definitions

A CPOG is a directed graph (V, E), whose *vertices* V and *arcs* $E \subseteq V \times V$ are labelled with Boolean functions, or *conditions*, $\phi : V \cup E \to (\{0,1\}^X \to \{0,1\})$, where $\{0,1\}^X \to \{0,1\}$ is a Boolean function on a set of Boolean *variables* X.

Fig. 3 (the top left box) shows an example of a CPOG H containing 4 vertices $V = \{a, b, c, d\}$, 6 arcs and 2 variables $X = \{x, y\}$. Vertex d is labelled with condition $x + y$ (i.e. 'x OR y'), arcs (b, c) and (c, b) are labelled with conditions x and y, respectively. All other vertices and arcs have trivial conditions 1 (trivial conditions are not shown for clarity); we call such vertices and arcs *unconditional*.

There are $2^{|X|}$ possible assignments of variables X, called *codes*. Each code induces a subgraph of the CPOG, whereby all the vertices and arcs, whose conditions evaluate to 0 are removed. For example, by assigning $x = y = 0$ one obtains graph H_{00} shown in the bottom right box in Fig. 3; vertex d and arcs (b, c) and (c, b) have been removed from the graph, because their conditions

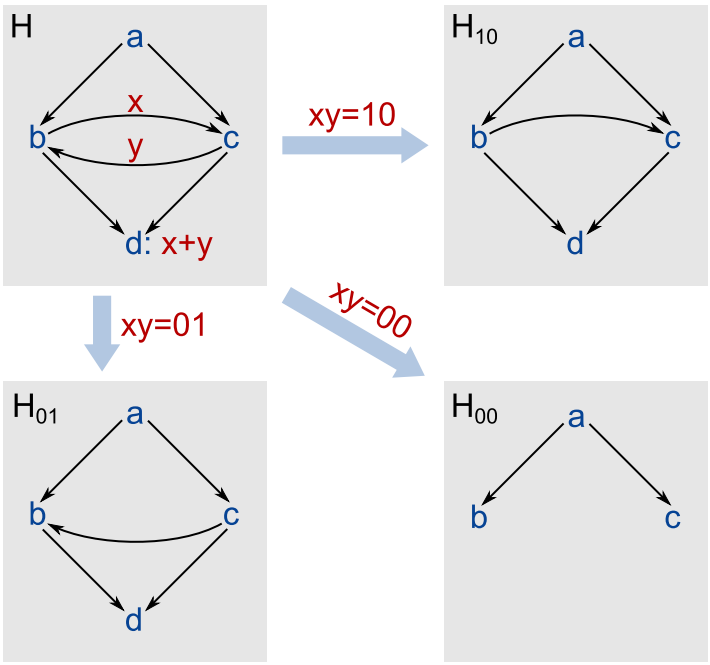


Figure 3: A CPOG and the associated family of graphs

are equal to 0 when $x = y = 0$. Different codes can produce different graphs, therefore a CPOG with $|X|$ variables can potentially specify a *family* of $2^{|X|}$ graphs. Fig. 3 shows two other members of the family specified by CPOG H: H_{01} and H_{10}, corresponding to codes 01 and 10, respectively, which differ only in the direction of the arc between vertices b and c. Codes will be denoted in a bold font, e.g. $\mathbf{x} = 01$, to distinguish them from vertices and variables.

It is often useful to focus only on a subset $C \subseteq \{0,1\}^X$ of codes, which are meaningful in some sense. For example, code 11 applied to CPOG H in Fig. 3 produces a graph with a loop between vertices b and c, which is undesirable if arcs are interpreted as causality. A Boolean *restriction function* $\rho : \{0,1\}^X \to \{0,1\}$ can be used to compactly specify the set $C = \{\mathbf{x} \mid \rho(\mathbf{x}) = 1\}$ and its complement $DC = \{\mathbf{x} \mid \rho(\mathbf{x}) = 0\}$, which are often referred to as the *care* and *don't care* sets [16]. By setting $\rho = \overline{xy}$ one can disallow the code $\mathbf{x} = 11$ as $\rho(11) = 0$, thereby restricting the family of graphs specified by CPOG H to three members only, which are all shown in Fig. 3.

The *size* $|H|$ of a CPOG $H = (V, E, X, \phi, \rho)$ is defined as:

$$|H| = |V| + |E| + |X| + \left| \bigcup_{z \in V \cup E} \phi(z) \cup \rho \right|,$$

where $|\{f_1, f_2, \ldots, f_n\}|$ stands for the size of the smallest circuit [17] that computes all Boolean functions in the set $\{f_1, f_2, \ldots, f_n\}$.

4.2 Families of partial orders

A CPOG $H = (V, E, X, \phi, \rho)$ is *well-formed* if every allowed code $\mathbf{x}$ produces an acyclic graph $H_{\mathbf{x}}$. By computing the transitive closure $H^*_{\mathbf{x}}$ one can obtain a *strict partial order*, an irreflexive and transitive relation on the set of *events* corresponding to vertices of $H_{\mathbf{x}}$.

We can therefore interpret a well-formed CPOG as a specification of a family of partial orders. We use the term *family* instead of the more general term *set* to emphasise the fact that partial orders are *encoded*, that is each partial order $H^*_{\mathbf{x}}$ is paired with the corresponding code $\mathbf{x}$. For example, the CPOG shown in Fig. 3 specifies the family comprising the partial order H^*_{00}, where event a precedes concurrent events b and c, and two total orders H^*_{01} and H^*_{10} corresponding to sequences *acbd* and *abcd*, respectively.

The *language* $\mathcal{L}(H)$ *of a CPOG* H is the set of all possible linearisations of partial orders contained in it. For example, the language of the CPOG shown in Fig. 3 is $\mathcal{L}(H) = \{abc, acb, abcd, acbd\}$. One of the limitations of the CPOG model is that it can only describe finite languages. However, this limitation is irrelevant for the purposes of this paper since event logs are always finite.

It has been demonstrated in [18] that CPOGs are a very efficient model for representing families of partial orders. In particular, they can be exponentially more compact than *Labelled Event Structures* [19] and *Petri Net unfoldings* [20]. Furthermore, for some applications CPOGs provide more comprehensible models than other widely used formalisms, such as *Finite State Machines* and *Petri Nets*, as has been shown in [4] and [5]. This motivated the authors to investigate the applicability of CPOGs to process mining.

4.3 Synthesis

In the previous sections we have demonstrated how one can extract partial orders from a given CPOG. However, the opposite problem is more interesting: *derive the smallest CPOG description for a given a set of partial orders*. This problem is called *CPOG synthesis* and it is an essential step in the proposed CPOG-based approach to process mining.

A number of CPOG synthesis methods have been proposed to date. The simplest method is based on graph colouring [4] and produces CPOGs with all conditions having at most one literal. Having at most one literal per condition is a serious limitation for many applications, but we found that the method works well for process mining. A more sophisticated approach, which produces CPOGs with more complex conditions has been proposed in [21], however, it has poor scalability and cannot be applied to large process mining instances. The most scalable approach to date, as confirmed by the experiments in §7.3, has been developed in [22] and is based on simulated annealing. All encoding methods are supported by open-source modelling framework WORKCRAFT [7], which we used in our experiments. In general, the CPOG synthesis problem is still in active research phase and new approximate methods are currently being developed. A promising direction for overcoming this challenge is based on reducing the CPOG synthesis problem to the problem of Finite State Machine synthesis [23].

5 From Event Logs to CPOGs

When visualising behaviour of an event log, it is difficult to identify a single technique that performs well for any given log due to the *representational bias* exhibited by existing process discovery algorithms. For example, if the event log describes a simple workflow behaviour, then the α-algorithm [9] is usually the best choice. However, if non-local dependencies are present in the behaviour, the α-algorithm will not be able to find them, and then other approaches, e.g. based on the theory of regions [12][24][25], may deliver best results. The latter techniques in turn are not robust when dealing with *noisy event logs*, for which other approaches may be more suitable [26][27]. There are many event logs for which none of the existing process discovery techniques seem to provide a satisfactory result according to the quality metrics presented in §3; for instance, see our simple motivating example in §2.

In this section we describe two approaches for translating a given event log L into a compact CPOG representation H. The first approach, which we call the *exact CPOG mining*, treats each trace as a totally ordered sequence of events and produces a CPOG H such that $L = \mathcal{L}(H)$. This approach does not introduce any new behaviours, hence the discovered models are *precise*.

The second approach attempts to exploit the concurrency between the events in order to discover *simpler* and *more general* models, hence we call it the *concurrency-aware CPOG mining*. This approach may in fact introduce new behaviours, which could be interpreted as new possible interleavings of the traces contained in the given event log L, hence producing a CPOG H that overapproximates the log, i.e. $L \subseteq \mathcal{L}(H)$. Both approaches satisfy the *fitness* criteria, that is, the discovered models cover all traces of the event log.

5.1 Exact CPOG mining

The *exact CPOG mining* problem is stated as follows: given an event log L, derive a CPOG H such that $L = \mathcal{L}(H)$. This can be trivially reduced to the CPOG synthesis problem. Indeed, each trace $t = e_1 e_2 \cdots e_m$ can be considered a total order of events $e_1 \prec e_2 \prec \cdots \prec e_m$. Therefore, a log $L = \{t_1, t_2, \cdots, t_n\}$ can be considered a set of n total orders and its CPOG representation can be readily obtained via CPOG synthesis. The solution always exists, but it is usually not unique. If uniqueness is desirable one can fix the assignment of codes to traces, in which case the result of synthesis can be presented in the *canonical form* [5].

For example, given event log $L = \{abcd, cdab, badc, dcba\}$ described in §2, the exact mining approach produces the CPOG shown in Fig. 1. As has already been discussed in §2, the resulting CPOG is very compact and provides a more comprehensible representation of the event log compared to conventional models used in process mining, such as Petri Nets or BPMNs.

When a given event log contains concurrency, the exact CPOG mining approach may lead to suboptimal results. For example, consider a simple event log $L = \{abcd, acbd\}$. If we directly synthesise a CPOG by treating each trace of this log as a total order, we will obtain the CPOG H shown in Fig. 4 (left). Although

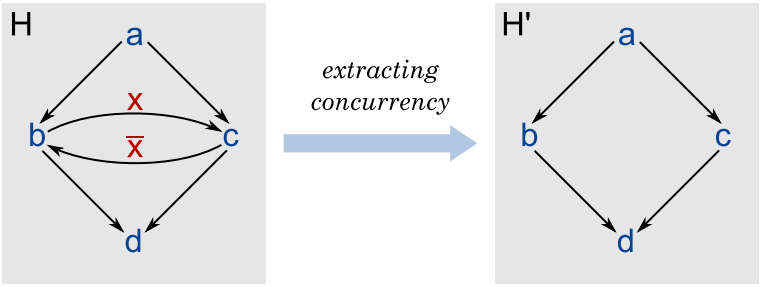


Figure 4: CPOG mining from event log $L = \{abcd, acbd\}$

$L = \mathcal{L}(H)$ as desired, the CPOG uses a redundant variable x to distinguish between the two total orders even though they are just two possible linearisations of the same partial order, where $a \prec b$, $a \prec c$, $b \prec d$, and $c \prec d$. It is desirable to recognise and extract the concurrency between events b and c, and use the information for simplifying the derived CPOG, as shown in Fig. 4 (right). Note that the simplified CPOG H' still preserves the language equality: $L = \mathcal{L}(H')$.

Since exact CPOG mining is a special case of the general CPOG synthesis problem (all given partial orders are in fact total orders), it is reasonable to expect that more efficient methods exist. The authors are unaware of such methods at present, but believe that this may be an interesting topic for research.

5.2 Concurrency-aware CPOG mining

This section presents an algorithm for extracting concurrency from a given event log and using this information for simplifying the result of the CPOG mining. Classic process mining techniques based on Petri Nets generally rely on the α-algorithm for concurrency extraction [1]. We introduce a new concurrency extraction algorithm, which differs from the classic α-algorithm in two aspects. On the one hand, it is more conservative when declaring two given events concurrent, which may lead to the discovery of more precise process models. On the other hand, it considers not only adjacent events in a trace as candidates for the concurrency relation but all event pairs, and therefore can find concurrent events even when the distance between them in traces is always greater than one, as we demonstrate below by an example. This method works particularly well in combination with CPOGs due to their compactness, however, we believe that it can also be useful in combination with other formalisms.

First, let us introduce convenient operations for extracting subsets of traces from a given event log L. Given an event e, the subset of L's traces containing e will be denoted as L_e, while the subset of L's traces not containing e will be denoted as $L_{\overline{e}}$. Clearly, $L_e \cup L_{\overline{e}} = L$. Similarly, given events e and f, the subset of L's traces containing both of them with e occurring before f will be denoted as $L_{e \to f}$. Note that $L_e \cap L_f = L_{e \to f} \cup L_{f \to e}$, i.e. if two events appear in a trace, they must be ordered one way or another. For instance, if $L = \{abcd, acbd, abce\}$ then $L_e = \{abce\}$, $L_{\overline{a}} = \emptyset$, $L_{a \to b} = L$, and $L_{a \to d} = \{abcd, acbd\}$. An event e is

conditional if $L_e \neq \emptyset$ and $L_e \neq L$, otherwise it is *unconditional*. A conditional event will necessarily have a non-trivial condition (neither 0 nor 1) in the mined CPOG. Similarly, a pair of events e and f is *conditionally ordered* if $L_{e \to f} \neq \emptyset$ and $L_{e \to f} \neq L_e \cap L_f$. Otherwise, e and f are *unconditionally ordered*.

We say that a conditional event r *indicates* the order between events e and f in an event log L if one of the following holds:

- $L_r \subseteq L_{e \to f}$
- $L_r \subseteq L_{f \to e}$
- $L_{\overline{r}} \subseteq L_{e \to f}$
- $L_{\overline{r}} \subseteq L_{f \to e}$

In other words, the existence or non-existence of the event r can be used as an indicator of the order between the events e and f. For example, if $L = \{abcd, acbd, abce\}$, then e indicates the order between b and c. Indeed, whenever we observe event e in a trace we can be sure that b occurs before c in that trace: $L_e \subseteq L_{b \to c}$. This notion leads to a simple concurrency oracle.

Definition 1 (Concurrency oracle). *Two events e and f are concurrent if they are conditionally ordered and no event r indicates their order.*

Intuitively, the order between two truly concurrent events should not be indicated by anything, i.e. it should have no side effects. Indeed, if one of the orderings is in any sense special and there is an indicator of this, then the events are not really concurrent, or at least they are *not always concurrent*. CPOGs are capable of expressing such *conditional concurrency* in a compact form. The *indicates relation* has been inspired by and is similar to the *reveals relation* from [28].

The above concurrency oracle is built on the simplest possible indicator – a single event whose occurrence happens to distinguish the order between two other events. We found this oracle to be very useful and efficient in practice, but it may be too weak in certain cases, in particular, similarly to the α-algorithm it declares all events concurrent in the motivation example from §2, resulting in a very imprecise process model $a \parallel b \parallel c \parallel d$. Fortunately, we can strengthen the oracle by using second-order relations between events as indicators.

We say that a pair of events (r, s) *indicates* the order between events e and f in an event log L if one of the following holds:

- $L_{r \to s} \subseteq L_{e \to f}$
- $L_{r \to s} \subseteq L_{f \to e}$

In other words, the order between the events r and s can be used as an indicator of the order between the events e and f. For example, if $L = \{abcd, cdab, badc, dcba\}$, then the order between events a and b indicates the order between events c and d (and vice versa). Indeed, whenever a occurs before b in a trace, we know that c occurs before d: $L_{a \to b} = L_{c \to d}$. We can use such second-order indicates relation for defining a more conservative concurrency oracle.

Definition 2 (Rank-2 concurrency oracle). *Two conditionally ordered events e and f are concurrent if (i) no event r indicates their order, and (ii) no pair of events (r, s) indicates their order.*

One can consider more sophisticated combinations of events and the order between them in the definition of the concurrency oracle, hence leading to a *hierarhcy of rank-N oracles*. Indeed, the order can be indicated by a triple $\{r, s, t\}$ of events, or a combination of an event r and an ordering $s \to t$, etc. A detailed investigation of the hierarchy of concurrency oracles is beyond the scope of this paper, but we believe that the hierarchy may be useful for choosing the right precision of the obtained models during process discovery.

The following example, suggested by an anonymous reviewer, highlights the difference between the proposed concurrency oracles and the α-algorithm.

Consider event log $L = \{xay_1y_2y_3bz, xby_1y_2y_3az, xy_1y_2y_3abz, xy_1y_2y_3baz\}$. The α-algorithm does not declare events a and y_2 concurrent, because they never appear adjacent in a trace (i.e. they are not in the so-called *directly-follows* relation). The proposed simple oracle however does declare them concurrent; in fact the whole chain $y_1 \prec y_2 \prec y_3$ is declared concurrent to both a and b, hence compressing the event log into one partial order $x \prec (a \parallel b \parallel y_1 \prec y_2 \prec y_3) \prec z$. The rank-2 oracle is very conservative in this example and does not declare any events concurrent; indeed, the ordering $a \to y_1$ is very rare (it appears only in the first trace) and can therefore be used as an indicator of $a \to b$, etc. The sensitivity of rank-N oracles to such rare combinations may be a disadvantage in some cases. To deal with this problem one can set a threshold for discarding rare indicators, a common approach when dealing with noisy event logs.

We are now ready to describe the algorithm for concurrency-aware CPOG mining. The algorithm takes an event log L as input and produces a CPOG H such that $L \subseteq \mathcal{L}(H)$.

1. Extract the concurrency: find all conditionally ordered pairs of events e and f, such that the order between them is not indicated by any events or pairs of events (when using the rank-2 oracle). Call the resulting set of concurrent pairs of events C.
2. Convert each trace $t \in L$ into a partial order p by relaxing the corresponding total order according to the set C. Call the resulting set of partial orders P.
3. Perform the exact CPOG synthesis on the obtained set of partial orders P to produce the resulting CPOG H.

Note that the resulting CPOG H indeed satisfies the condition $L \subseteq \mathcal{L}(H)$, since we can only add new linearisations into H in step (2) of the algorithm, when we relax a total order corresponding to a particular trace by discarding some of the order relations.

Let us now apply the algorithm to the previous examples. Given log $L = \{abcd, cdab, badc, dcba\}$ from §2, the algorithm does not find any concurrent pairs, because the order between each pair of events is indicated by the order between the complementary pair of events (e.g. $L_{a\to b} = L_{c\to d}$). Hence, $C = \emptyset$ and the result of the algorithm coincides with the exact CPOG mining, as shown in §2. Given log $L = \{abcd, acbd\}$ from §5.1, the algorithm finds one pair of concurrent events, namely (b, c), which results in collapsing of both traces of L into the same partial order with trivial CPOG representation shown in Fig. 4 (right).

6 From Control Flow to Data

As demonstrated in the previous section, one can derive a compact CPOG representation from a given event log using CPOG mining techniques. The obtained representations however rely on opaque Boolean variables, which make the result difficult to comprehend. For example, Fig. 1(a) provides no intuition on how a particular variable assignment can be interpreted with respect to the process under observation. The goal of this section is to present a method for the automated extraction of useful data labels from a given event log (in particular from available event attributes) and using these labels for constructing 'transparent' and easy-to-comprehend predicates, which can substitute the opaque Boolean variables. This is similar to the application of conventional machine learning techniques for learning 'decision points' in process models or in general for the automated enhancement of a given model by leveraging the available data present in the event log [1].

More formally, given an event log L and the corresponding CPOG H our goal is to explain how a particular condition f can be interpreted using data available in L. Note that f can be as simple as just a single literal $x \in X$ (e.g. the arc $a \to b$ in Fig. 1(a)), in which case our goal is to explain a particular Boolean variable; however, the technique introduced in this section is applicable to any Boolean function of the CPOG variables $f : \{0,1\}^X \to \{0,1\}$, in particular, one can use the technique for explaining what the restriction function ρ corresponds to in the process, effectively discovering the process *invariants*. We achieve the goal by constructing an appropriate instance of the *classification problem* [29].

Let $n = |E|$ be the number of different events in L, and k be the number of different event attributes available in L. Remember that attributes of an event e can be accessed via function $attr(e)$, see §3. Hence, every event e in the log defines a *feature vector* $\hat{e}$ of dimension k where the value at i-th position corresponds to the value of the i-th attribute[4] of e. For instance, the feature vector $\hat{e}_1$ corresponding to the event e_1 in Table 1 is ("10-04-2015 9:08am", 25.0, 17, Low). Some features may need to be abstracted before applying the technique described below to produce better results, e.g. timestamps may be mapped to five discrete classes: *morning*, *noon*, *afternoon*, *evening* and *night*.

Feature vectors	Class
$\{\hat{e} \mid e \in \sigma \wedge \sigma \in L_f\}$	True
$\{\hat{e} \mid e \in \sigma \wedge \sigma \in L_{\overline{f}}\}$	False

Table 2: Binary classification problem for function f and event log L.

The key observation for the proposed method is that all traces in the log L can be split into two disjoint sets, or *classes*, with respect to the given function f: i) set L_f, containing the traces where f evaluates to 1, and ii) set $L_{\overline{f}}$ containing the traces where f evaluates to 0. This immediately leads to an instance of the *binary classification problem* on n feature vectors, as illustrated in Table 2. In

[4] We assume a total order on the set of event attributes.

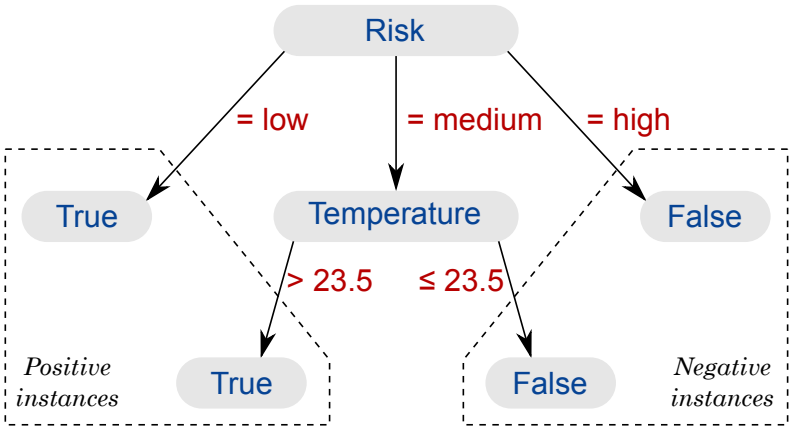


Figure 5: Decision tree built for function $f = x$ in the CPOG of Fig. 1(a).

other words, every event belonging to a trace where the function f evaluates to 1 is considered to belong to the class we learn, that is, the class labelled as True in Table 2 (the remaining events do not belong to this class). Several methods can be applied to solve this problem, including *decision trees* [30], *support vector machines* [31], and others. In this work we focus on decision trees as they provide a convenient way to extract predicates defined on event attributes, which can be directly used for substituting opaque CPOG conditions. The method is best explained by way of an example.

Consider the event log in Table 1, which contains a few data attributes for each event. The traces underlying the log are $\{abcd, cdab, badc, dcba\}$. Fig. 1(a) shows the corresponding CPOG produced by the CPOG mining techniques presented in the previous section. Let us try to find an interpretation of the variable x by applying the above procedure with $f = x$. The set L_f equals to $L_{a \to b}$, i.e. it contains traces 1 and 2, wherein event a occurs before event b and therefore $f = 1$. Therefore, feature vectors $\hat{e}_1$-$\hat{e}_8$ provide the positive instances of the class to learn (the first eight events of the log belong to traces 1 and 2), while feature vectors $\hat{e}_9$-$\hat{e}_{16}$ provide the negative ones. The decision tree shown in Fig. 5 is a possible classifier for this function, which has been derived automatically using machine learning software Weka [32]. By combining the paths in the tree that lead to positively classified instances, one can derive the following predicate for f: $risk = low \vee (risk = medium \wedge temperature > 23.5)$. This predicate can be used to substitute the opaque variable x in the mined CPOG.

One can use the same procedure for deriving the explanation for all variables and/or conditions in a CPOG, thereby providing a much more comprehensible representation for the event log. Note that for complementary functions, taking the negation of the classification description will suffice, e.g. $\overline{x}$ in Fig. 1(a) can be substituted with predicate $risk \neq low \wedge (risk \neq medium \vee temperature \leq 23.5)$. Alternatively, one can derive the predicate for a complementary function by combining paths leading to the negative instances; for example, for $f = \overline{x}$ the resulting predicate is $risk = high \vee (risk = medium \wedge temperature \leq 23.5)$.

The learned classifier can be tested for evaluating the quality of representation of the learned concept. If the quality is unacceptable then the corresponding condition may be left unexplained in the CPOG. Therefore in general the data extraction procedure may lead to partial results when the process contains concepts which are 'difficult to learn'. For example, in the discussed case study the condition $f = y$ could not be classified exactly.

A coarse-grain alternative to the technique discussed in this section is to focus on *case attributes* instead of event attributes. Case attributes are attributes associated with a case (i.e., a trace) as a whole instead to individual events [1]. Furthermore, the two approaches can be combined with the aim of improving the quality of obtained classifiers.

7 Tool support and experiments

The techniques presented in this paper have been implemented as a plugin for the Workcraft framework [7][33], which is a collection of opensource tools for design, verification and analysis of concurrent systems. In this section we will describe our backend tools, frontend capabilities, and will analyse the performance of the current implementation on a set of realistic process mining benchmarks.

7.1 Backend tools

We rely on three backend tools: PGminer [8], Scenco [34] and Weka [35].

PGminer is a contribution of this paper, developed specifically for the efficient concurrency-aware mining of CPOGs from event logs as described in §5.2. It can handle event logs with multiple occurrences of an event in a trace, by splitting such traces into scenarios that are free from event repetitions, which is essential for our current implementation (this is further discussed in §7.2). An important feature of the tool is that the results are represented in an algebraic form using the algebra of Parameterised Graphs introduced in [5] (hence the name, PGminer). This avoids the quadratic explosion of the representation due to transitive arcs appearing after the concurrency extraction step. PGminer has been implemented as a process mining library written in Haskell [36] and can be run as a standalone command line tool or via the Workcraft frontend.

Scenco is a collection of CPOG synthesis algorithms that have been developed in a series of publications and integrated in Workcraft: graph colouring based single literal synthesis [4], SAT-based synthesis [21], and heuristic synthesis [22]. We use Scenco for encoding collections of partial orders produced by PGminer. As discussed in §7.3, CPOG synthesis is the main bottleneck of the current process mining implementation. Our future work will be dedicated to the development of a custom CPOG synthesis algorithm specialised for collections of partial orders obtained from process logs after concurrency extraction.

Weka is a collection of opensource machine learning and data mining algorithms. In the current workflow Weka is used for extracting meaningful conditions from event log data, as discussed in §6. Our future work includes integration of Weka into Workcraft for better interoperability with other methods.

7.2 Details of current implementation

WORKCRAFT [7][33] is a collection of software tools united by a common modelling infrastructure and a graphical user interface. WORKCRAFT suppors several interpreted graph models: Petri Nets, Finite State Machines, digital circuits, dataflow structures, xMAS communication networks, and CPOGs, the latter being particularly important for this work. It provides a unified frontend for visual editing and simulation of interpreted graph models, as well as facilities for processing these models by established model-checking and synthesis tools.

WORKCRAFT features a plugin for CPOGs, providing an interface which allows a user to create and edit CPOGs by using a graphical editor, or by describing graphs algebraically using the algebra of parameterised graphs [5]. It is possible to convert between the graphical and algebraic representations automatically.

The authors developed a *process mining plugin* for WORKCRAFT that provides the functionality for importing event logs, manipulating them in the graphical editor, performing concurrency extraction using PGMINER, and synthesising compact CPOG models using SCENCO.

An event log can be imported either directly, in which case each trace is treated as a total order of events, or indirectly via PGMINER, in which case the log undergoes the concurrency extraction procedure leading to a more compact representation and allowing for handling bigger event logs. The current implementation treats multiple occurrences of the same event as different events, e.g., trace (a, b, a, b, c, b, c) is interpreted as $(a_1, b_1, a_2, b_2, c_1, b_3, c_2)$. This can have a negative impact on the concurrency extraction procedure; to avoid this PGMINER provides a method for spliting traces into scenarios which are free from repeated events. For the example at hand this leads to splitting the trace into three sub-traces (a, b), (a, b, c), and (b, c), i.e. whenever a current sub-trace cannot be extended without repeating an event, a new sub-trace is started.

A collection of partial orders can be synthesised into a compact CPOG model using the SCENCO plugin. Our experiments have shown that only heuristic CPOG synthesis [22] can cope with event logs of realistic sizes. Other, more sophisticated encoding methods are not scalable enough. Once a CPOG representation of an event log is obtained, the user can analyse it visually and investigate the meaning of encoding variables using the CPOG projection functionality provided in WORKCRAFT or by performing data mining in WEKA.

7.3 Experiments

Table 3 summarises the experimental results. All benchmark logs come from the process mining community: artificial logs derived from the simulation of a process model (Caise2014, BigLog1, Log1, Log2), a real-life log containning the jobs sent to a copy machine (DigitalCopier), a software log (softwarelog), and real-life logs in different other contexts [37] (documentflow, incidenttelco, purchasetopay, svn_log, telecom). Some of the logs are challenging even for prominent process mining software, and they were therefore chosen as a realistic challenge for testing the capabilities of the developed tools. Note that the '# events' column reports the number of different events after cyclic traces are split by PGMINER.

Benchmark	Log parameters				Tool runtime				CPOG size		
	File size	# traces	# events	# partial orders	Direct import	Indirect import	Concurrency extraction	CPOG encoding	# arcs	# vars	# gates
BigLog1-100	21Kb	100	22	16	<1 sec	<1 sec	<1 sec	1 sec	33	5	103
BigLog1-500	102Kb	500	22	27	3 sec	<1 sec	<1 sec	1 sec	37	5	174
BigLog1-1000	204Kb	1000	22	26	6 sec	<1 sec	<1 sec	2 sec	37	5	149
Caise2014	25Kb	100	40	401	2 sec	88 sec	1 sec	-	-	-	-
softwarelog	4Kb	5	210	167	<1 sec	1 sec	<1 sec	19 sec	464	8	1751
DigitalCopier-300	70Kb	300	33	15	9 sec	<1 sec	<1 sec	2 sec	37	4	56
DigitalCopier	173Kb	750	33	9	35 sec	<1 sec	<1 sec	1 sec	45	4	78
DigitalCopierMod	116Kb	1000	15	6	4 sec	<1 sec	<1 sec	1 sec	18	3	1
documentflow	208Kb	12391	70	651	2 min	11 sec	1 sec	-	-	-	-
incidenttelco-100	17Kb	100	20	25	<1 sec	<1 sec	<1 sec	4 sec	61	5	225
incidenttelco	161Kb	956	22	77	8 sec	1 sec	<1 sec	8 sec	97	7	641
Log1-filtered	3.6Mb	5000	47	402	-	3 min	13 sec	-	-	-	-
Log2	2Mb	10000	22	32	8.25 min	1 sec	1 sec	1 sec	38	5	194
purchasetopay	232Kb	10487	21	20	3.7 min	1 sec	<1 sec	1 sec	34	5	140
svn_log	24Kb	765	13	92	3 sec	1 sec	<1 sec	2 sec	69	7	581
telecom	15Kb	1000	38	122	2 sec	1 sec	<1 sec	4 sec	194	7	937

Table 3: Summary of experimental results

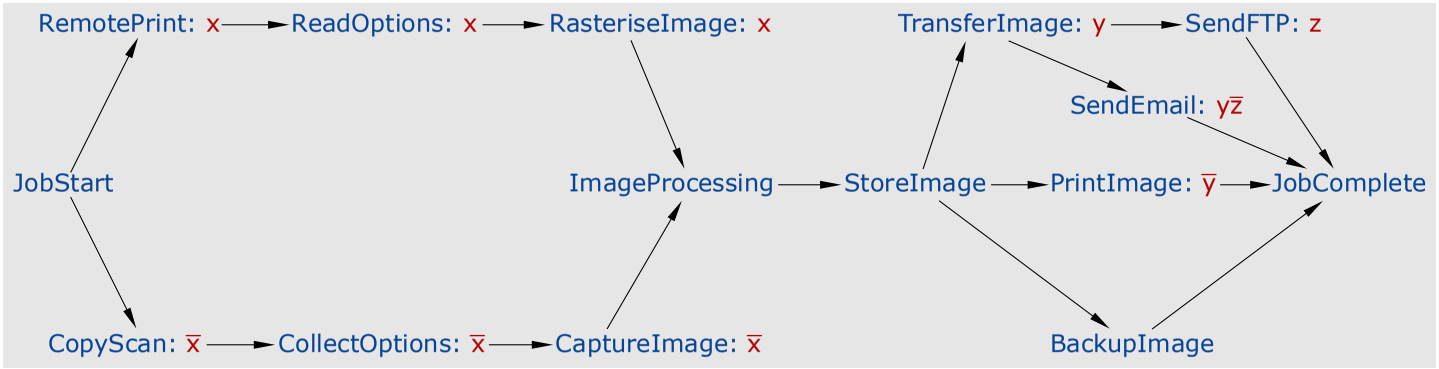
As can be seen from the table, there are normally a lot more traces than partial orders, thanks to the successful concurrency extraction by PGminer. However, two cases, namely Caise2014 and softwarelog, are exceptions: they contain traces with particularly many event repetitions which leads to a significant increase of the logs due to the log splitting heuristic described in §7.2.

The experiments show that PGminer, when used as a standalone command line tool, is very scalable and can efficiently handle most logs (see column 'Concurrency extraction'). Indeed, most execution times are below or around 1 second, and only Log1-filtered (a 3.6Mb log) takes 13 seconds to be processed.

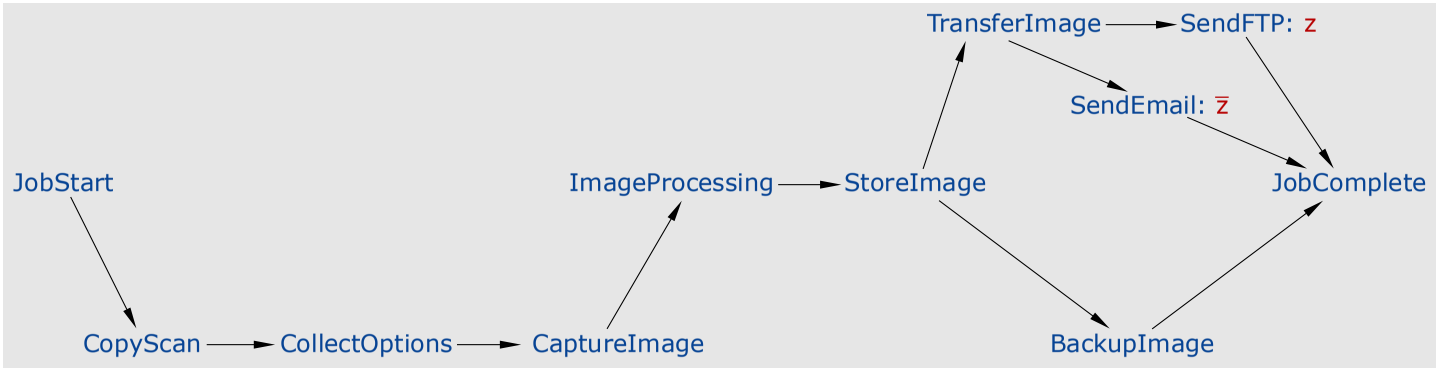
Workcraft is less scalable, as one would expect from a feature-rich graphical editor. Direct import of some logs takes minutes and Log1-filtered cannot be directly imported at all. Indirect import of logs, which is performed by invoking PGminer first, is more scalable: all logs can be imported this way with most execution times being around 1 second.

CPOG synthesis is the bottleneck of the presented process mining approach. It is a hard computational problem and even heuristic solutions do not currently scale well; in particular, cases with more than 200 partial orders could not be handled. Note that synthesised CPOGs are typically sparse; the number of vertices $|V|$ coincides with the number of events in a log, and as can be seen from the table, the number of arcs $|E|$ in resulting CPOGs is often close to $|V|$. The sparseness of synthesised CPOGs should be exploited by future synthesis tools.

Fig. 6 shows an example of a mined model for the DigitalCopierMod log, which is a modified version of DigitalCopier: a new event *BackupImage* was added to demonstrate concurrency extraction, while multiple occurrences of other events were eliminated. The CPOG model was produced by Workcraft from a log containing 1000 traces in under 1 second; note that we manually improved the layout and added colours to enhance the readability. The bottom subfigure shows a projection obtained by setting $x = 0$ and $y = 1$. One can easily compute CPOG projections with Workcraft when exploring CPOG process models.



(a) Process model. Note the restriction function $\rho = z \Rightarrow y$, i.e. setting $y = 0$ and $z = 1$ is forbidden.



(b) Projection of the process model with $x = 0$ and $y = 1$.

Figure 6: Visualisation example: CPOG models of the DigitalCopierMod log.

8 Related work and discussion

Process mining is a vibrant research field and there are a few relevant research works that are worth discussing and comparing with the proposed CPOG-based representation of event logs. [38] is very close to our work in spirit: it convincingly advocates for using event structures as a unified representation of process models and event logs. As has been recently shown in [18], CPOGs can be exponentially more compact than event structures, therefore we believe that the approach presented in [38] can benefit from the extra compactness provided by CPOGs.

The authors of [39] introduce *trace alignment*, a technique for aligning traces of an event log thereby producing a better visual representation. It uses a matrix representation where rows correspond to traces and columns correspond to positions within each trace. Trace alignment is a powerful visualisation technique that aims to maximise the consensus of the event positions across different traces. In contrast to CPOGs, trace alignment does not compress the information encountered in the traces, nor does it provide a bridge between the control flow and data as proposed in this paper. Furthermore, the trace alignment matrix is a final process mining representation, whilst CPOGs are intended as an intermediate representation and can be used to algebraically operate on event logs.

Another relevant research direction [40][41] relies on the notion of partially-ordered event data and introduces techniques for conformance checking of this type of event representations. In particular, [40] presents the notion of partially-ordered trace (*p-trace*). As in the case of CPOGs, a p-trace allows for explicit concurrency between events of the same trace. P-traces can be computed by careful inspection of the event timestamps. The techniques to extract p-traces are extended in [41] in order to deal with data. However, the use of data attributes is narrower compared to the approach presented in this paper: data attributes are split into read/write accesses to data values, and simple rules to extract concurrency and dependency are introduced to take into account the role of a data access within a trace. We believe that the techniques for relating control flow and data presented in this paper may be applied in the scope of [40][41].

As discussed in the previous section, several challenges need to be faced before the presented techniques can be adopted in industrial process mining solutions, e.g. the complexity of CPOG synthesis algorithms, the fine-tuning of parameters of the data mining techniques, and some others. Due to the inability of CPOGs to directly represent cyclic behavior, we currently only focus on using CPOGs for visualisation and as an intermediate representation of event logs, which can be further transformed into an appropriate process mining formalism, such as Petri Nets or BPMNs. Although some syntactic transformations already exist to transform CPOGs into contextual Petri Nets [33], we believe that finding new methods for discovery of process mining models from CPOGs is an interesting direction for future research.

Another future research direction is to consider CPOGs as compact algebraic objects that can be used to efficiently manipulate and compare event logs [5]. Since a CPOG corresponding to an event log can be exponentially smaller, this

may help to alleviate the memory requirements bottleneck for current process mining tools that store 'unpacked' event logs in memory.

Event logs are not the only suitable input for the techniques presented in this paper: we see an interesting link with the work on *discovery of frequent episodes*, e.g. as reported recently in [42]. *Episodes* are partially ordered collections of events (not activities), and as such they can also be represented by CPOGs. This may help to compress the information provided by frequent episodes, especially if one takes into account the fact that current algorithms may extract a large number of episodes, which then need to be visualised for human understanding.

9 Conclusions

This paper describes the first steps towards the use of CPOGs in the field of process mining. In particular, the paper presented the automatic derivation of the control flow part of the CPOG representation from a given event log, and then the incorporation of meta data contained in the log as conditions of the CPOG vertices and arcs. We have implemented most of the reported techniques and some preliminary experiments have been carried out.

The future work includes addressing the challenges described in the previous section, as well as an evaluation of how derived CPOGs can be useful in practice for understanding event data.

Acknowledgments. The authors would like to thank Alessandro de Gennaro and Danil Sokolov for their help with the integration of the developed process mining tools into Workcraft. Many organisations supported this research work: Andrey Mokhov was supported by Royal Society Research Grant 'Computation Alive' and EPSRC project UNCOVER (EP/K001698/1); Josep Carmona was partially supported by funds from the Spanish Ministry for Economy and Competitiveness (MINECO) and the European Union (FEDER funds) under grant COMMAS (ref. TIN2013-46181-C2-1-R); Jonathan Beaumont is currently a PhD student sponsored by a scholarship from the School of Electrical and Electronic Engineering, Newcastle University, UK.

References

1. Wil van der Aalst. *Process Mining: Discovery, Conformance and Enhancement of Business Processes.* Springer, 2011.
2. The ProM framework homepage. http://www.promtools.org/, 2010.
3. Minseok Song and Wil MP van der Aalst. Supporting process mining by showing events at a glance. *Proc. of Annual Workshop on Information Technologies and Systems (WITS)*, pages 139–145, 2007.
4. Andrey Mokhov. *Conditional Partial Order Graphs.* PhD thesis, Newcastle University, 2009.
5. Andrey Mokhov and Victor Khomenko. Algebra of Parameterised Graphs. *ACM Transactions on Embedded Computing Systems (TECS)*, 13(4s):143, 2014.

6. Andrey Mokhov and Josep Carmona. Event Log Visualisation with Conditional Partial Order Graphs: from Control Flow to Data. In *Proceedings of the International Workshop on Algorithms & Theories for the Analysis of Event Data, ATAED 2015, Brussels, Belgium, June 22-23, 2015.*, pages 16–30, 2015.
7. The WORKCRAFT framework homepage. http://www.workcraft.org, 2009.
8. The PGMINER tool repository. https://github.com/tuura/process-mining, 2015.
9. Wil M. P. van der Aalst, Ton Weijters, and Laura Maruster. Workflow mining: Discovering process models from event logs. *IEEE TKDE*, 16(9):1128–1142, 2004.
10. Joos C. A. M. Buijs, Boudewijn F. van Dongen, and Wil M. P. van der Aalst. A genetic algorithm for discovering process trees. In *Proceedings of the IEEE Congress on Evolutionary Computation, CEC 2012, Brisbane, Australia, June 10-15, 2012*, pages 1–8, 2012.
11. Sander J. J. Leemans, Dirk Fahland, and Wil M. P. van der Aalst. Discovering block-structured process models from event logs - A constructive approach. In *Application and Theory of Petri Nets and Concurrency - 34th International Conference, PETRI NETS 2013, Milan, Italy, June 24-28, 2013. Proceedings*, pages 311–329, 2013.
12. Josep Carmona, Jordi Cortadella, and Michael Kishinevsky. New region-based algorithms for deriving bounded Petri nets. *IEEE Trans. Computers*, 59(3):371–384, 2010.
13. Robin Bergenthum, Jörg Desel, Robert Lorenz, and Sebastian Mauser. Synthesis of Petri nets from finite partial languages. *Fundam. Inform.*, 88(4):437–468, 2008.
14. Andrey Mokhov, Danil Sokolov, and Alex Yakovlev. Adapting asynchronous circuits to operating conditions by logic parametrisation. In *IEEE International Symposium on Asynchronous Circuits and Systems (ASYNC)*, pages 17–24. IEEE, 2012.
15. Andrey Mokhov, Maxim Rykunov, Danil Sokolov, and Alex Yakovlev. Design of processors with reconfigurable microarchitecture. *Journal of Low Power Electronics and Applications*, 4(1):26–43, 2014.
16. Giovanni de Micheli. *Synthesis and Optimization of Digital Circuits*. McGraw-Hill Higher Education, 1994.
17. Ingo Wegener. *The Complexity of Boolean Functions*. Johann Wolfgang Goethe-Universitat, 1987.
18. Hernán Ponce De León and Andrey Mokhov. Building bridges between sets of partial orders. In *International Conference on Language and Automata Theory and Applications (LATA)*, 2015.
19. Mogens Nielsen, Gordon Plotkin, and Glynn Winskel. Petri nets, event structures and domains, part I. *Theoretical Computer Science*, 13:85–108, 1981.
20. Kenneth McMillan. Using unfoldings to avoid the state explosion problem in the verification of asynchronous circuits. In *Proceedings of Computer Aided Verification conference (CAV)*, volume 663, page 164, 1992.
21. Andrey Mokhov, Arseniy Alekseyev, and Alex Yakovlev. Encoding of processor instruction sets with explicit concurrency control. *Computers & Digital Techniques, IET*, 5(6):427–439, 2011.
22. Alessandro de Gennaro, Paulius Stankaitis, and Andrey Mokhov. A heuristic algorithm for deriving compact models of processor instruction sets. In *International Conference on Application of Concurrency to System Design (ACSD)*, 2015.
23. Tiziano Villa, Timothy Kam, Robert K Brayton, and Alberto L Sangiovanni-Vincentelli. *Synthesis of finite state machines: logic optimization*. Springer Publishing Company, Incorporated, 2012.

24. Marc Solé and Josep Carmona. Light region-based techniques for process discovery. *Fundam. Inform.*, 113(3-4):343–376, 2011.
25. Jan Martijn E. M. van der Werf, Boudewijn F. van Dongen, Cor A. J. Hurkens, and Alexander Serebrenik. Process discovery using integer linear programming. In *ATPN*, pages 368–387, 2008.
26. Christian W. Günther and Wil M. P. van der Aalst. Fuzzy mining - adaptive process simplification based on multi-perspective metrics. In Gustavo Alonso, Peter Dadam, and Michael Rosemann, editors, *BPM*, volume 4714 of *Lecture Notes in Computer Science*, pages 328–343. Springer, 2007.
27. A.J.M.M. Weijters, W.M.P. van der Aalst, and A.K. Alves de Medeiros. Process mining with the heuristics miner-algorithm. Technical Report WP 166, BETA Working Paper Series, Eindhoven University of Technology, 2006.
28. Stefan Haar, Christian Kern, and Stefan Schwoon. Computing the reveals relation in occurrence nets. *Theor. Comput. Sci.*, 493:66–79, 2013.
29. Tom M. Mitchell. *Machine learning*. McGraw Hill series in computer science. McGraw-Hill, 1997.
30. J. Ross Quinlan. Induction of decision trees. *Machine Learning*, 1(1):81–106, 1986.
31. Corinna Cortes and Vladimir Vapnik. Support-vector networks. *Machine Learning*, 20(3):273–297, 1995.
32. Mark Hall, Eibe Frank, Geoffrey Holmes, Bernhard Pfahringer, Peter Reutemann, and Ian H Witten. The WEKA data mining software: an update. *ACM SIGKDD explorations newsletter*, 11(1):10–18, 2009.
33. Ivan Poliakov, Danil Sokolov, and Andrey Mokhov. Workcraft: a static data flow structure editing, visualisation and analysis tool. In *Petri Nets and Other Models of Concurrency–ICATPN 2007*, pages 505–514. Springer, 2007.
34. The Scenco tool website. http://www.workcraft.org/scenco, 2015.
35. The Weka tool website. http://www.cs.waikato.ac.nz/ml/weka, 2015.
36. Simon Marlow et al. Haskell 2010 language report. *Available online http://www.haskell.org/*, 2010.
37. ActiTraC: Active Trace Clustering. http://www.processmining.be/actitrac/, 2014.
38. Marlon Dumas and Luciano García-Bañuelos. Process mining reloaded: Event structures as a unified representation of process models and event logs. In *Application and Theory of Petri Nets and Concurrency*, pages 33–48. Springer, 2015.
39. R. P. Jagadeesh Chandra Bose and Wil M. P. van der Aalst. Process diagnostics using trace alignment: Opportunities, issues, and challenges. *Inf. Syst.*, 37(2):117–141, 2012.
40. Xixi Lu, Dirk Fahland, and Wil M. P. van der Aalst. Conformance checking based on partially ordered event data. In *Business Process Management Workshops - BPM 2014 International Workshops, Eindhoven, The Netherlands, September 7-8, 2014, Revised Papers*, pages 75–88, 2014.
41. Xixi Lu, Ronny Mans, Dirk Fahland, and Wil M. P. van der Aalst. Conformance checking in healthcare based on partially ordered event data. In *Proceedings of the 2014 IEEE Emerging Technology and Factory Automation, ETFA 2014, Barcelona, Spain, September 16-19, 2014*, pages 1–8, 2014.
42. Maikel Leemans and Wil M. P. van der Aalst. Discovery of frequent episodes in event logs. In *Proceedings of the 4th International Symposium on Data-driven Process Discovery and Analysis (SIMPDA 2014), Milan, Italy, November 19-21, 2014.*, pages 31–45, 2014.